Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Informatica

Soluzione di vincoli XCSP mediante Gecode

Relatore:
Prof. Maurizio Gabbrielli

Tesi di laurea di:
Alessio Riccardo

Sessione II
Anno Accademico 2010/2011

Dedico questa tesi a coloro che
hanno sempre creduto in me.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

In questa tesi si è testato Gecode, un risolutore di vincoli, su dati rappresentati in un formato (XCSP) non nativamente supportato da Gecode, attraverso un plug-in chiamato *x4g* [16]. Successivamente i risultati ottenuti con Gecode sono stati confrontati con altri due risolutori: Mistral e Abscon, i quali accettano il formato di rappresentazione XCSP di un problema di soddisfacimento di vincoli. Infine si è testata la possibilità di utilizzo di uno scheduler di risolutori per rendere più efficiente la ricerca di soluzioni di problemi di soddisfacimento di vincoli.

Un risolutore di vincoli (solver) è un programma che dato in input un problema di soddisfacimento di vincoli (CSP), ne determina la soddisfacibilità, ovvero elabora la soluzione (se esiste) dell'insieme di vincoli di cui il problema è composto. Un CSP è descritto da un insieme di variabili che possono assumere valori appartenenti ad uno specifico dominio ed un insieme di vincoli che mettono in relazione variabili e valori assumibili da esse. I problemi di soddisfacimento vincoli vengono impiegati in numerose attività reali:

- Problemi di allocazione di risorse.
- Problemi di scheduling.

- Gestione e configurazione di reti di telecomunicazioni.
- Riconoscimeto di immagini e focalizzazione di attenzione in sistemi di visione.
- Progettazione di circuiti digitali.
- Realizzazione di assistenti intelligenti per applicazioni web.

Una tecnica per ottimizzare la risoluzione di tali problemi è quella suggerita da un approccio a portfolio [4]. Tale tecnica, usata anche in ambiti come quelli economici, prevede la combinazione di più risolutori i quali assieme possono generare risultati migliori di un approccio a singolo risolutore. Allo scopo di ottenere migliori risultati da un insieme di solver piuttosto che da uno singolo è opportuno l'uso di algoritmi di apprendimento automatico [5] (noto in letteratura come *Machine Learning*). L'apprendimento automatico rappresenta una delle aree fondamentali dell'intelligenza artificiale e si occupa della realizzazione di sistemi e algoritmi capaci di sintetizzare nuova conoscenza a partire dall'osservazione dell'ambiente e dall'esperienza. Per il nostro intento, l'ampliamento di conoscenza concerne il sapere in anticipo la quantità di risorse che un solver impiega a risolvere un problema di vincoli. In questo contesto, osservare l'ambiente vorrebbe dire considerare la struttura logica di un CSP, cosa non banale, e riconoscere la relazione che questa struttura può avere con il metodo di ricerca delle soluzioni da parte dei solver presi in esame. Avendo a che fare con stime misurabili (tempo, uso della cpu, ecc.), sembra più idoneo cercare di apprendere dall'esperienza, che in questo caso è rappresentata dalla raccolta di dati da test sui risolutori. A questo punto è indicato l'uso di un classificatore, ovvero una funzione di inferenza che dovrebbe essere in grado di predire il corretto valore di output per ogni input valido. Per fare ciò si è avvalsi in primo luogo di WEKA, un software open-source scritto in Java, che

mette a disposizione una collezione di algoritmi di machine learning. Successivamente con una versione modificata di CPHydra [6], un portfolio di constraint solver sviluppato presso la University College of Cork (UCC), si è sperimentato un approccio case based reasoning, una popolare tecnica di machine learning. Finendo per confrontare le due tecniche (classificazione e case based reasoning) applicate ai dati raccolti.

Il resto di questa tesi è così articolato: nel secondo capitolo vengono descritti i problemi di soddisfacimento di vincoli ed i concetti fondamentali sull'apprendimento automatico, mostrandone anche qualche esempio tipico; nel terzo capitolo viene presentata la programmazione con vincoli, introdotti i risolutori Gecode, Mistral e Abscon, il formalismo di modellazione XCSP e il plug-in *x4g*; nel quarto capitolo si entra in merito ai test per la raccolta dati e alle tecniche di apprendimento automatico, i risultati del lavoro vengono considerati nel quinto capitolo, per giungere alle conclusioni nel sesto capitolo.

Capitolo 2

Concetti preliminari

In questo capitolo saranno introdotti il concetto di apprendimento automatico ed i Problemi di Soddisfacimento di Vincoli (Constraint Satisfaction Problems, CSP), che rappresentano i tipici problemi analizzati e risolti dai linguaggi di Programmazione a Vincoli (Constraint Programming, CP).

2.1 Problemi di soddisfacimento di vincoli

Molti problemi nell'ambito dell'Intelligenza Artificiale sono classificabili come CSP [1]; fra questi citiamo problemi di complessità combinatorica, di allocazione di risorse, pianificazione e ragionamento temporale. In generale, le attività tipiche del paradigma CSP sono computazionalmente intrattabili (NP-hard).

Un problema di soddisfacimento di vincoli è definito da una tripla $\langle X,D,C\rangle$ dove:

- X è un insieme di variabili, $\{X_1, ..., X_n\}$.
- D è un insieme di domini, $\{D_1, ..., D_n\}$, uno per ogni variabile.
- C è un insieme di vincoli che specificano combinazioni di valori ammesse.

Ogni dominio D_i è costituito da un insieme di valori ammessi $\{v_1, ..., v_k\}$ per la variabile X_i. Ogni vincolo C_i è costituito da una coppia $\langle$*ambito,rel*$\rangle$ dove *ambito* è una tupla di variabili che partecipano nel vincolo e *rel* è una relazione che definisce i valori che tali variabili possono assumere [2]. Una relazione può essere rappresentata come elenco esplicito di tutte le tuple di valori che soddisfano il vincolo, o come una relazione astratta che supporta due operazioni: controllare se una tupla è membro della relazione ed enumerare i membri stessi. Ad esempio se X_1 ed X_2 hanno entrambe il dominio $\{A, B\}$ allora il vincolo che afferma che le due variabili devono avere valori diversi si può scrivere come $\langle(X_1, X_2), [(A, B), (B, A)]\rangle$ o come $\langle(X_1, X_2), X_1 \neq X_2\rangle$.

Per risolvere un problema di soddisfacimento di vincoli, dobbiamo definire uno spazio degli stati e il concetto di soluzione. Ogni stato in un CSP è definito dall'**assegnamento** di valori ad alcune o tutte le variabili, $\{X_i = v_i, X_j = v_j, ...\}$. Un assegnamento che non viola alcun vincolo è chiamato **consistente** o legale. Un assegnamento è **completo** se a tutte le variabili è assegnato un valore; una **soluzione** di un CSP è un assegnamento completo e consistente. Un assegnamento è **parziale** se menziona solo alcune delle variabili. Dato un CSP si potrebbe voler trovare:

- solo una soluzione (caso ovvio se il problema ne ammette una sola),
- tutte le soluzioni possibili,
- la migliore soluzione, avendo a disposizione una funzione obiettivo definita su alcune o tutte le variabili.

2.1.1 Consistenza

Il concetto di consistenza locale rappresenta un CSP come un grafo dove ogni variabile è rappresentata con un nodo e ogni vincolo binario tra variabili è rappresentato come un arco, il processo di forzare la consistenza locale in ogni

parte del grafo causa l'eliminazione dei valori inconsistenti nel grafo stesso. Di seguito vengono descritti i diversi tipi di consistenza:

Node-Consistency: consistenza di grado 1. Un nodo di un grafo è consistente se, per ogni valore nel dominio, i vincoli unari associati al nodo sono soddisfatti. Tutti i valori che non soddisfano i vincoli possono sicuramente essere eliminati dal dominio. Un grafo viene definito node-consistent se tutti i suoi nodi sono consistenti. Se, per esempio, in un CSP avessimo una variabile X con dominio iniziale [0..10], e il vincolo $X > 2$, per rendere il grafo node-consistent è necessario eliminare i valori [0..2] dal dominio di X.

Arc-Consistency: consistenza di grado 2. La consistenza di grado 2 si ottiene partendo da un grafo node-consistent verificando la consistenza di tutti gli archi. Un arco è consistente se per ogni valore nel dominio di una variabile esiste almeno un valore nel dominio dell'altra variabile che soddisfi i vincoli associati all'arco. Ovviamente la rimozione di alcuni valori a causa dell'arc-consistency rende necessarie ulteriori verifiche che coinvolgano i vincoli unari sulle due variabili in gioco. Questo procedimento iterativo deve essere ripetuto fino a che la rete non converge ad uno stato stabile arc-consistent. La sua generalizzazione a vincoli non binari prende il nome di **Hyper Arc Consistency**.

Path-Consistency: consistenza di cammino. La consistenza di cammino restringe i vincoli binari utilizzando vincoli impliciti che sono inferiti considerando triplette di variabili. Un insieme di due variabili $\{X_i, X_j\}$ è cammino path-consistente rispetto a una terza variabile X_m se, per ogni assegnamento $\{X_i = a, X_j = b\}$ consistente con i vincoli su $\{X_i, X_j\}$, esiste un assegnamento per X_m che soddisfa i vincoli su $\{X_i, X_m\}$ e $\{X_m, X_j\}$. Si parla di consistenza di cammino perchè si può pensare di considerare un cammino da X_i a X_j con X_m nel mezzo.

K-Consistency: consistenza di grado k. Dato un grafo consistente fino al grado $k-1$, la consistenza di grado k si ottiene scegliendo ogni possibile $(k-1)$-

pla di variabili, consistente per definizione con i vincoli imposti, e cercando un valore per ogni ulteriore variabile del problema che soddisfi i vincoli fra tutte le k variabili prese in considerazione.

Si dimostra che, se un grafo contenente n variabili è k-consistent con $k < n$, allora per trovare una soluzione è sufficiente una ricerca nello spazio restante. In un grafo di n variabili n-consistent, la soluzione può quindi essere trovata senza eseguire alcuna ricerca, basta avere l'accortezza di scegliere nel giusto ordine le variabili da assegnare. Tuttavia, rendere un grafo n-consistent ha complessità esponenziale in n. Occorre quindi, anche in questo caso, ricercare il valore di k che fornisca il miglior trade-off tra la complessità del processo iterativo di consistenza e la riduzione effettuata sullo spazio di ricerca.

2.1.2 Tecniche di risoluzione

I CSP con domini finiti vengono generalmente risolti utilizzando una forma di ricerca. Le tecniche più utilizzate sono varianti del backtracking, propagazione di vincoli, e ricerca locale. Molti algoritmi per risolvere CSP effettuano una ricerca sistematica tra i possibili assegnamenti di valori alle variabili del problema. Sono algoritmi che garantiscono di trovare una soluzione, se esiste, o di provare che il problema è insolvibile. Il principale svantaggio di questi algoritmi è che potrebbero impiegare molto tempo per dare una risposta. Il più comune tra questa famiglia di algoritmi è il **Backtracking**, un algoritmo ricorsivo che mantiene un assegnamento parziale delle variabili. La ricerca con backtracking indica una ricerca in profondità, dove inizialmente nessuna variabile è assegnata. Ad ogni passo, una variabile è scelta e, a turno, tutti i possibili valori sono assegnati ad essa. Per ogni valore, viene controllata la consistenza dell'assegnamento parziale con i vincoli, in caso di coerenza, una chiamata ricorsiva viene eseguita. Quando tutti i valori di una variabile sono stati esaminati senza successo l'algoritmo effettua il backtracking, tornando indietro nell'albero di ricerca

fino a trovare una strada ancora inesplorata. Esistono diverse varianti del backtracking. **Backmarking** permette di evitare alcune ridondanze sui controlli dei vincoli e nella scoperta di inconsistenze. **Backjumping** permette di ridurre lo spazio di ricerca tornando indietro in caso di fallimento alla variabile più recente tra quelle legate da vincoli con la variabile che ha causato il fallimento, questo può far saltare all'indietro l'algoritmo di più di un livello alla volta.

Le tecniche di propagazione di vincoli sono utilizzate per modificare un CSP. Più precisamente, si tratta di metodi che applicano una forma di consistenza locale. La propagazione dei vincoli ha vari usi. In primo luogo, trasformare un problema in uno equivalente ma di solito più semplice da risolvere. In secondo luogo, rivelare la soddisfacibilità o insoddisfacibilità del problema. Il più popolare metodo di propagazione di vincoli è l'algoritmo **AC-3**, che impone la consistenza d'arco. Tra gli algoritmi di propagazione solitamente implementati troviamo il **Forward Checking** e il **Look Ahead**. Il primo, dopo ogni assegnamento, elimina i valori incompatibili con quello appena istanziato dai domini delle variabili non ancora istanziate. Se ad un certo punto della computazione uno o più domini risultassero vuoti, il meccanismo fallirebbe subito e proverebbe altri valori per le variabili già istanziate, evitando così di scendere in un ramo dell'albero decisionale che porterebbe sicuramente al fallimento e, di conseguenza, al backtracking. L'algoritmo **Look Ahead** prevede che ad ogni istanziazione venga eseguito il **Forward Checking**, e venga inoltre controllata l'esistenza, nei domini delle variabili ancora libere, di valori compatibili con i vincoli contenenti solo variabili non istanziate. In altre parole, si controlla se, a causa della riduzione effettuata dal **Forward Checking**, i valori rimasti nei domini delle variabili libere possano ancora portare, scendendo nell'albero, ad una soluzione, o se siano viceversa destinati ad un sicuro fallimento.

I metodi di ricerca locale sono algoritmi soddisfacibilità incompleti. Possono trovare una soluzione di un problema, ma potrebbero fallire anche se il

problema è soddisfacibile. Funzionano migliorando iterativamente un asegnamento completo sulle variabili. Ad ogni passo, un piccolo numero di variabili cambia valore, con l'obiettivo generale di aumentare il numero di vincoli soddisfatti dall'assegnamento. L'algoritmo **Min-Conflicts** è un algoritmo di ricerca locale che assegna valori casuali a tutte le variabili di un CSP, seleziona casualmente le variabili e assegna ad ognuna il valore che ha il numero minimo di conflitti. Infine, l'**apprendimento dei vincoli** è una delle più importanti tecniche utilizzate dai moderni risolutori di CSP per ottenere l'efficienza su problemi complessi. L'idea nell'apprendimento dei vincoli è di trovare un insieme minimo delle variabili dell'insieme dei conflitti (l'insieme dei conflitti per una variabile X è l'insieme delle variabili precedentemente assegnate che sono collegate a X da vincoli) che causa il problema, aggiungendo un vincolo al CSP riguardante questo insieme di variabili e i corrispondenti valori, evitando così di ripercorrere strade che portano sicuramente a un fallimento.

2.1.3 Ordinamento di variabili e valori

Una funzione *euristica*, applicata ad un nodo n, valuta la distanza, quindi lo sforzo computazionale, che separa il nodo n dalla soluzione. Ordinando le strade aperte dalla più "promettente" alla meno "promettente", cioè da quella che ha una funzione euristica migliore a quella peggiore, è possibile raggiungere una buona soluzione in tempi ragionevoli anche per i problemi più complessi. Ovviamente, affinché un'euristica porti ad una soluzione, occorre che essa sia buona, sia in grado cioè di ordinare il più esattamente possibile le strade, quindi i rami dell'albero di ricerca, secondo la probabilità che possano portare ad una soluzione, o alla soluzione ottima. Gli algoritmi di ricerca per CSP richiedono sia un ordine in cui le variabili del problema vengono selezionate, sia un ordine di scelta dei valori da assegnare. La scelta dei giusti ordinamenti può migliorare considerevolmente l'efficienza della risoluzione dei vincoli.

Le variabili possono avere un ordinamento statico o dinamico. L'ordinamento statico, come il nome suggerisce, viene definito prima che la ricerca abbia inizio e non viene cambiato. Nell'ordinamento dinamico invece la scelta di quale variabile istanziare dipende dallo stato corrente della ricerca. L'ordinamento dinamico non è però utilizzabile con tutti gli algoritmi di ricerca, ad esempio nel **Backtracking** semplice nessuna informazione extra è disponibile durante la ricerca per cambiare l'ordine iniziale delle variabili. Mentre, nel **Forward Checking**, lo stato corrente include i domini delle variabili per come sono stati potati dal corrente insieme di istanziazioni, è così possibile basare la scelta della prossima variabile su questa informazione.

Diverse euristiche sono state sviluppate e analizzate per la selezione dell'ordine delle variabili. La più comune è la **MRV** (*Minimum Remaining Values*), basata sul principio del "first-fail" che può essere espresso come "Per avere successo, prova prima dove hai più possibilità di fallire", essa suggerisce di scegliere la variabile con il minor numero di valori legali rimanenti. In questo modo, si prova a tagliare il prima possibile i rami dell'albero di ricerca che non portano a nessuna soluzione. Se tutte le variabili hanno lo stesso numero di valori possibili, l'**euristica di grado**, sceglie la variabile che partecipa a più vincoli con le variabili ancora non assegnate, seguendo il principio di gestire per primi i casi più difficili cerca di ridurre il fattore di ramificazione delle scelte future. Esiste anche un'euristica per l'ordinamento statico delle variabili che è adatto al **Backtracking** semplice. Questa euristica dice: scegli la variabile con il più vasto numero di vincoli con le variabili precedentemente istanziate. Per esempio, durante la risoluzione del problema della colorazione di un grafo, è ragionevole assegnare un colore al vertice con achi comuni a vertici già colorati, così da poter determinare un eventuale conflitto il più presto possibile.

Dopo avere scelto quale variabile istanziare, si deve decidere quale valore assegnare ad essa. Di nuovo, l'ordine di valutazione dei valori può avere un

considerevole impatto sul tempo di ricerca della soluzione. Comunque, se tutte le soluzioni sono richieste o non ci sono soluzioni, allora l'ordinamento dei valori è indifferente. Un diverso ordinamento dei valori riorganizza i rami dell'albero di ricerca, assicurando che se un ramo porta a una soluzione verrà percorso prima di rami che portano ad un fallimento. Per esempio, se un CSP ha soluzione, e se un valore corretto viene scelto per ogni variabile, allora la soluzione sarà trovata senza backtracking. Supponiamo di aver scelto una variabile da istanziare: come possiamo scegliere quale valore provare per primo? Può accadere che nessun valore rispetta i vincoli, si dovranno considerare quindi tutti i valori e l'ordine non avrà importanza. Altrimenti può in certi casi essere efficace l'euristica del **valore meno vincolante** che predilige il valore che lascia più libertà di scelta alle variabili adiacenti sul grafo dei vincoli, lasciando quindi massima flessibilità ai successivi assegnamenti di variabili.

2.1.4 Problemi giocattolo

Esistono una varietà di problemi, cosiddetti giocattolo, risolvibili agevolmente con la programmazione con vincoli. Tra questi i problemi delle N-Regine, della colorazione di una mappa e di crypto-aritmetica hanno una posizione privilegiata.

N-Regine

Il rompicapo delle N-Regine [17] consiste nel trovare il modo posizionare N-Regine (pezzo degli scacchi) su una scacchiera $N * N$ tali che nessuna di esse possa catturarne un'altra, usando i movimenti standard della regina. Perciò, una soluzione dovrà prevedere che nessuna regina abbia una colonna, traversa o diagonale in comune con un'altra regina. Un tipico modo di modellare il problema è di assumere che ogni regina si trovi in una colonna diversa dalle altre e di assegnare a una variabile R_i (con dominio 1...N) alla regina nella

i-esima colonna indicando la posizione della regina nella riga. Il vincolo che esprime la non attaccabilità tra ogni coppia di regine può esprimersi con la seguente forma:

$$i \neq j \rightarrow (R_i \neq R_j \ \& \ |i - j| \neq |R_i - R_i|)$$

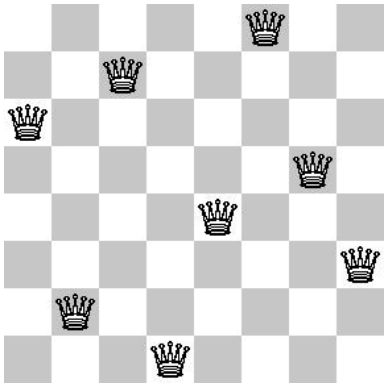

Figura 2.1: Configurazione di una scacchiera con 8 regine.

Map Coloring

Un altro problema che è spesso utilizzato per dimostrare il potenziale della programmazione con vincoli e per spiegare concetti e algoritmi per CSP è il problema della colorazione di una mappa o *Map Coloring* [18]. Consiste nello stabilire se sia possibile colorare una mappa geografica utilizzando solo un numero definito e limitato di colori dove due paesi confinanti non possono avere lo stesso colore. Questo problema è conosciuto nel campo della teoria dei grafi come **k-colorability**. Molti problemi di allocazione di risorse possono essere ricondotti a questo problema. Un esempio di questo è il problema dell'assegnamento delle frequenze radio, un problema di telecomunicazione dove l'obiettivo è quello di assegnare frequenze radio alle stazioni in modo che tutte possano operare contemporaneamente senza udibili interferenze. Utilizzando la teoria dei grafi, la mappa geografica può essere astratta come un grafo con un nodo

per ogni paese e un arco congiungente ogni coppia di paesi confinanti. Dato un grafo di grandezza arbitraria, il problema è stabilire quando i nodi possono essere colorati utilizzando solo k colori in modo che ogni coppia di nodi adiacente non abbia lo stesso colore. Il problema di k-colorability dei grafi è formulato come un CSP dove ogni nodo nel grafo è una variabile e il dominio associato alle variabili sono i possibili colori, mentre l'insieme delle condizioni imposte sulle coppie di vertici adiacenti è l'insieme dei vincoli.

Crypto-aritmetica

Un altro esempio di problemi giocattolo sono i giochi enigmistici di crypto-aritmetica. In questi giochi ogni lettera corrisponde a una cifra diversa e il vincolo da rispettare è la formula matematica espressa con le lettere stesse. Questo è un tipico esempio di problema di crypto-aritmetica:

```
SEND + MORE = MONEY
```

Ogni lettera viene identificata con una variabile (con dominio 0...9), rescrivendo direttamente la formula con in equivalente vincolo aritmetico:

```
             1000*S + 100*E + 10*N + D
+            1000*M + 100*O + 10*R + E
= 10000*M + 1000*O + 100*N + 10*E + Y
```

e aggiungendo in vincolo $S \neq 0$ e $M \neq 0$.

2.2 Apprendimento Automatico

Il Machine Learning [5] (o Apprendimento Automatico) è il settore dell'Itelligenza Artificiale che si occupa di realizzare dispositivi artificiali capaci di emulare le modalità di ragionare tipiche dell'uomo: riconoscere, decidere, scegliere.

L'obiettivo del Machine Learning è far migliorare un sistema senza bisogno del continuo intervento umano. Per fare questo tali sistemi devono essere in grado di apprendere, ovvero di estrarre informazioni su un determinato problema esaminando una serie di esempi ad esso relativi, chiamati complessivamente training set. Un metodo per risolvere, anche se molto parzialmente, questo problema è dotare le macchine simboliche di capacità di ragionamento induttivo oltre che deduttivo. Si tratta, in sostanza, di fornire dei meccanismi tramite i quali le macchine possano imparare con esempi. È il tipico processo tramite il quale formuliamo delle generalizzazioni a partire da alcuni esempi particolari. Il ragionamento induttivo, infatti, procede da asserzioni singolari riguardanti particolari fatti o fenomeni (esempi) ad asserzioni universali esprimibili mediante ipotesi o teorie che spieghino i fatti dati e siano in grado di predirne di nuovi. La messa a punto di dispositivi artificiali capaci di apprendere riveste una tale importanza che costituisce uno degli obiettivi primari di diversi settori scientifici: la Statistica, l'Intelligenza Artificiale, le Scienze Cognitive, ecc.

Gli algoritmi di apprendimento automatico sono tradizionalmente divisi in tre principali tipologie:

Apprendimento supervisionato

Nell'apprendimento supervisionato il training set è composto da una coppia di esempi determinata da un oggetto di input (tipicamente un vettore) e un valore di output desiderato (anche chiamato supervisory signal). Un algoritmo di apprendimento supervisionato genera una funzione di inferenza (classificatore) che dovrebbe essere in grado di predire il corretto valore di output per ogni input valido.

Apprendimento non supervisionato

In cui il problema diventa quello di trovare strutture nascoste in strutture dati non preclassificate da cui non è possibile valutare una possibile soluzione. L'apprendimento non supervisionato è strettamente collegato al problema di stima di densità in statistica.

Apprendimento con rinforzo

Questa tecnica di programmazione si basa sul presupposto di potere ricevere degli stimoli dall'esterno a seconda delle scelte dell'algoritmo. Gli algoritmi per il reinforcement learning tentano di determinare una politica tesa a massimizzare gli incentivi cumulati ricevuti dall'agente nel corso della sua esplorazione del problema. L'apprendimento con rinforzo differisce da quello supervisionato poiché non sono mai presentate delle coppie input-output di esempi noti, né si procede alla correzione esplicita di azioni subottimali.

Per quanto riguarda l'Apprendimento supervisionato tre sono le classi di problemi che possono essere ulteriormente caratterizzati, a seconda dei valori assunti dalla uscita y del sistema:

- Problemi di riconoscimento, se l'uscita è binaria ($y \in 0, 1$). Es. diagnosi medica di una determinata patologia.
- Problemi di classificazione, se y varia entro un insieme finito di valori non ordinati ($y \in 1, 2, ...m$). Es. riconoscimento di caratteri manoscritti.
- Problemi di regressione, se l'uscita è continua ($y \in R$). Es. previsione dell'andamento di serie temporali.

Il principale criterio di valutazione per un algoritmo di apprendimento è la sua capacità di generalizzazione, cioè di costruire ipotesi che predìcano correttamente il valore della funzione per esempi non appartenenti al training set. Per

valutare correttamente la capacità di generalizzazione di un'ipotesi si applica il metodo seguente:

- Si colleziona un insieme di esempi.
- Si divide tale insieme in due sottoinsiemi disgiunti: training set e test set.
- Si applica l'algoritmo di apprendimento al training set, ottenendo un'ipotesi h.
- Nei problemi di classificazione, si calcola la percentuale di esempi del test set correttamente classificati da h (accuratezza); nei problemi di regressione, si calcola una misura della "vicinanza" tra i valori predetti dall'ipotesi h per gli esempi del test set e quelli corretti.

La capacità di generalizzazione di un'ipotesi dipende anche dal particolare training set usato. Una stima statisticamente più significativa della capacità di generalizzazione richiede l'uso di training set diversi. Una tecnica sviluppata per questo scopo è la *cross-validation*:

- Si suddivide il training set iniziale in k sottoinsiemi disgiunti e complementari (tipicamente, k = 10).
- Si costruiscono k ipotesi, per ognuna delle quali si usano come training set k–1 sottoinsiemi, e si valuta la capacità di generalizzazione sul restante sottoinsieme.
- Si calcola l'accuratezza media delle k ipotesi.

2.2.1 I classificatori

Un sistema di classificazione o di riconoscimento, considerato in senso ampio, ha il compito di fornire ad un utente (uomo o calcolatore) una valutazione della

realtà fisica osservata e tale valutazione si avvale di una suddivisione della realtà (costituita da oggetti detti campioni o "pattern") in insiemi, aventi caratteristiche omogenee, detti classi. Nell'analisi di classificazione è importante valutare l'affidabilità del modello per fini predittivi, ciò viene definito mediante il calcolo dell'accuratezza del classificatore attraverso vari metodi. L'obiettivo dell'analisi di classificazione è la verifica dell'esistenza di differenze tra le classi in funzione delle variabili considerate e la formulazione di un modello che sia in grado di assegnare ciascun campione alla classe cui esso appartiene.

Esistono vari tipi di modelli di classificazione e differiscono per il formalismo utilizzato per rappresentare la funzione di classificazione. Segue un elenco di alcuni modelli di classificazione:

- Basati sugli esempi (Es. Nearest neighbor), memorizzano tutti gli esempi del training set ed assegnano la classe ad un oggetto valutando la "somiglianza" con gli esempi memorizzati (la cui classe è nota).

- Matematici (Es. Reti Neurali Artificiali, SVM), la funzione di classificazione è una funzione matematica, di cui si memorizzano i vari parametri.

- Statistici (Es. Naive Bayes), memorizzano i parametri delle varie distribuzioni di probabilità relative alle classi ed agli attributi per classificare un generico oggetto si possono stimare le probabilità di appartenenza alle varie classi.

- Logici (Es. Alberi e Regole di Decisione), la funzione di classificazione è espressa mediante condizioni logiche sui valori degli attributi.

Capitolo 3

Programmazione con vincoli

Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.[1]

La programmazione con vincoli [3] è un potente paradigma di programmazione usato da ricercatori ed esperti in numerosi campi come l'Intelligenza Artificiale, la Ricerca Operativa, i Linguaggi di Programmazione, il Calcolo Simbolico ecc. L'uso sistematico dei vincoli nella programmazione è iniziato negli anni '80. Recenti applicazioni di questo paradigma includono la computer grafica (per esprimere coerenza geometrica nell'analisi delle scene), elaborazione del linguaggio naturale (costruzione di parser efficienti), basi di dati (per verificare e/o ripristinare la consistenza sui dati), problemi di ricerca operativa, biologia molecolare, ingegneria elettronica, progettazione di circuiti e altro.

L'approccio su cui si fonda la programmazione a vincoli è cercare uno "stato" del "mondo" in cui un grande numero di vincoli sono contemporaneamente soddisfatti. Si assume che risolvere un problema **equivalga** a trovare un **mon-**

[1] Eugene C. Freuder, Inaugural issue of the Constraints Journal, 1997.

do possibile descritto da un numero (inizialmente sconosciuto) di variabili. Il programma va alla ricerca dei valori che, attribuiti alle variabili, meglio definiscono il "mondo" soggetto ai vincoli imposti. I vincoli differiscono dalle primitive normalmente definite dagli altri linguaggi di programmazione per il fatto che non specificano azioni singole da eseguire passo passo, ma piuttosto si limitano a specificare le proprietà di cui deve essere dotata la soluzione da trovare. Ma cosa è esattamente un vincolo? Un vincolo è semplicemente una relazione logica tra variabili, ognuna delle quali prende valori da un dominio dato. Un vincolo quindi restringe i possibili valori che la variabile può prendere, rappresenta un'informazione parziale sulle variabili d'interesse. I vincoli da applicare possono essere forniti embedded nel linguaggio di programmazione, oppure in librerie separate. Le programmazione a vincoli iniziò come programmazione logica a vincoli, introducendo vincoli integrati in un programma di tipo logico. Questa variante della programmazione logica fu opera di Jaffar e Lassez, che, nel 1987, svilupparono una classe di vincoli specificatamente progettata per essere usata nel linguaggio Prolog II [19]. Attualmente esistono molti interpreti per programmi logici a vincoli, come per esempio GNU Prolog [21]. A differenza dalla programmazione logica, i vincoli possono essere inseriti nella programmazione funzionale, nella riscrittura e nei linguaggi imperativi. Nella programmazione funzionale i vincoli sono implementati ad esempio nel linguaggio di programmazione multi-paradigma Oz [20]. Vincoli sono embedded (integrati) nel linguaggio imperativo Kaleidoscope. Nei linguaggi imperativi, tuttavia, i vincoli sono implementati principalmente mediante i cosiddetti constraint solving toolkits, che sono librerie separate, fornite insieme al linguaggio. ILOG CP Optimizer [23], è un esempio è di queste librerie per C++, Java e .NET. In questo lavoro di tesi sono stati presi in considerazione tre librerie per la programmazione basata su vincoli, Gecode, Mistral e Abscon, che verranno descritte nelle prossime sezioni. La programmazione a vincoli temporal concurrent

(TCC) e la programmazione a vincoli temporal concurrent non-deterministica (NTCC) sono varianti della programmazione a vincoli in cui entra in gioco la variabile tempo.

Un risolutore di vincoli è una procedura che trasforma un CSP T in uno equisoddisfacibile (o equivalente). Si dice:

- Completo se, dato un problema T, lo trasforma in un CSP o in una disgiunzione finita di CSP ad esso equisoddisfacibile, e tale che da ciascuno dei disgiunti sia immediato trarre ogni sua soluzione; se P è inconsistente, viene restituito fail.

- Incompleto se non è completo. Intuitivamente, dato un problema T, lo trasforma in un CSP più semplice ma non ancora abbastanza semplice.

Lo stato dell'arte nel mondo dei risolutori di vincoli è principalmente rappresentato dalla Constraint Solver Competition (CSC) [22]. Durante la scrittura di questa tesi l'evento ha raggiunto la sua quarta edizione internazionel nel 2009. Tuttavia, a questa competizione possono partecipare solo solver capaci di accettare un CSP nello specifico formato XML: XCSP [12]. I risultati vengono ordinati basandosi sul numero di istanze risolte dai solver e i pareggi vengono spezzati considerando il tempo totale minimo di risoluzione. Basandosi su questo criterio, la figura 3.1 mostra i risultati della competizione del 2009.

Tra i partecipanti PCS, Abscon, Choco, Sugar e Mistral sono vincenti in almeno una categoria. Nello specifico Mistral e Sugar sono appaiati vincendo 8 categorie su 21, poi Choco e Abscon con 2 categorie e PCS con 1 categoria. Sfortunatamente nessuna classifica sulla valocità dei solver o sul rapporto tra CPU time e numero di istanze risolte è fornita.

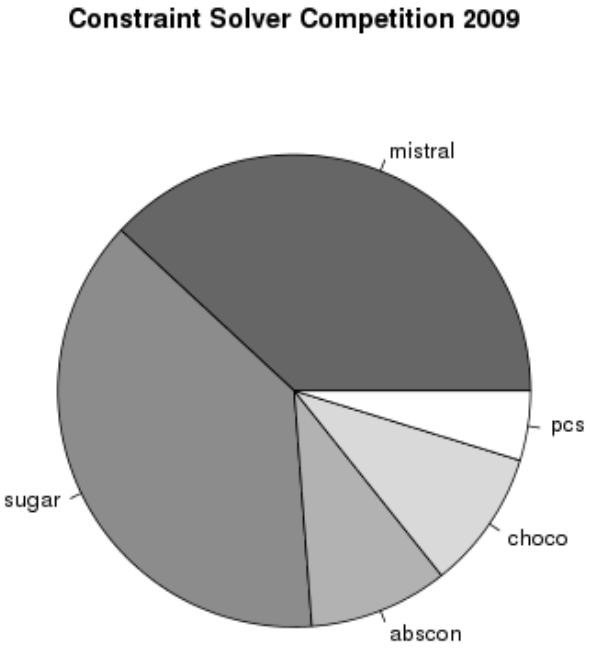

Figura 3.1: Vincitori della CSC del 2009.

3.1 Gecode

Gecode (Generic Constraint Development Environment) [14] è un risolutore di vincoli capace di performance allo stato dell'arte pur essendo modulare ed estensibile. Viene implementato come libreria C++ e distribuito come software free sotto licenza MIT, rendendolo portabile sulla maggior parte delle piattaforme. Supporta la programmazione di nuovi propagatori (implementazioni dei vincoli), strategie di branch e motori di ricerca. Nuovi domini possono essere implementati allo stesso livello di efficienza dei domini finiti e di variabili di insiemi di interi che sono predefiniti in Gecode. Gecode offre eccellenti prestazioni a runtime e di memoria. A titolo di esempio, Gecode ha vinto tutte le sfide MiniZinc [26] finora (in tutte le categorie): 2010, 2009 e 2008.

La ricerca in Gecode coinvolge due tecniche: *branching* e *exploration*. Il *branching* definisce la forma dell'albero di ricerca, mentre *exploration* specifi-

ca la strategia di esplorazione di parti dell'albero di ricerca e come modificare possibilmente la forma dell'albero di ricerca durante l'esplorazione (per esempio durante la ricerca della soluzione migliore con la strategia *branch-and-bound* aggiungendo nuovi vincoli). Le euristiche di scelta delle variabili e dei valori sono specificate nel *branching*, l'insieme delle possibili euristiche è fornito da Gecode. Si tratta in generale di euristiche standard, come ad esempio l'euristica di scelta della variabile più vincolata, con dominio minimo. Interessante è la scelta della variabile che tiene conto dell'Accomulated Failure Count (AFC), ovvero la somma di tutti i fallimenti incontrati durante la propagazione dei vincoli che dipendono da una variabile più la quantità di vincoli legati alla variabile (per dare un buon valore iniziale agli AFC). AFC è anche conosciuto come l'euristica di grado pesata di una variabile. Durante la valutazione di quale variabile istanziare è possibile imbattersi in casi in cui due o più variabili sono equivalenti per l'euristica, in questi casi in Gecode entra in gioco il *tie-breaking*, cioè un secondo criterio di valutazione per scegliere quale delle variabili scegliere. Il comportamento di default per il *tie-breaking* è la scelta della variabile che occupa la posizione più bassa nell'array delle variabili candidate, che in molte situazioni non è sufficiente per ottenere buoni risultati. Un possibile esempio di *tie-breaking* è la combinazione delle euristiche di grado e di dominio minimo. Con l'euristica di grado viene circoscritto un sottoinsieme di variabili con cardinalità di vincoli minima rispetto alle altre, se questo sottoinsieme non è un composto da una sola variabile allora viene scelto un ulteriore sottoinsieme di variabili con dominio minimo, se anche questa volta il sottoinsieme conta più di una variabile di dovrà ricorrere al comportamento di default del *tie-breaking*. I motori di ricerca predefiniti in Gecode sono tre: il classico *depth-first left-most* (**DFS**), il *branch-and-bound* (**BAB**) e il *depth-first left-most restart* (**Restart**). Mentre il DFS non può garantire di trovare la migliore soluzione, BAB continua la ricerca quando una soluzione è stata trovata aggiungendo un vincolo per

trovare una soluzione migliore in tutti i rimanenti nodi dell'albero di ricerca. Restart fa ripartire la ricerca quando il vincolo per la migliore soluzione viene aggiunto al nodo radice. Un altro aspetto caratterizzante della ricerca in Gecode è il *recomputation*. Il *recomputation* ricorda cosa è avvenuto durante il *branching*: invece che memorizzare un intero clone dello stato, solo la quantità di informazione necessaria a ripristinare gli effetti di un *brancher* è immagazzinata. Questa informazione è chiamata *choice* in Gecode. Ripristinare uno stato passato può servire se una strada presa da un *brancher* non porta a nessuna soluzione, o se una soluzione viene trovata ma sono richieste tutte le soluzioni.

Gecode può beneficiare del parallelismo durante la ricerca per cercare di renderla più efficiente. Viene utilizzata una architettura standard chiamata *work-stealing*, secondo la quale diversi thread esplorano parti differenti dell'albero di ricerca in parallelo. Inizialmente tutto il lavoro (l'intero albero da esplorare) è assegnato a un singolo thread, rendendolo occupato. Tutti gli altri thread inizialmente inattivi cercano di rubare lavoro dal singolo thread impegnato, rubare lavoro significa che una parte dell'albero di ricerca viene assegnato ad un thread che diventa così occupato. Durante tutta la ricerca se un thread occupato finisce il suo lavoro cercherà di rubarne altro da un thread lavoratore. Dato che il *work-stealing* è indeterministico (dipende dalla schedulazione dei thread, il carico di lavoro della macchina, e altri fattori), il lavoro che viene rubato varia tra differenti esecuzioni dello stesso problema: un thread inattivo può potenzialmente rubare diversi sottoalberi da diversi thread tra una esecuzione e l'altra. Differenti sottoalberi possono contenere diverse soluzioni, quindi è indeterministca anche quale soluzione è trovata per prima.

Sono state sviluppate e sono in via di sviluppo interfacce per Gecode, come l'interfaccia per FlatZinc[2] che permette a Gecode di elaborare modelli scritti in un linguaggio a basso livello per problemi con vincoli disegnato per essere facil-

[2]Linguaggio di modellazione a basso livello per problemi di vincoli

mente interfacciabile per risolutori di vincoli; ma anche interfacce per linguaggi come Ruby, Python, Prolog e Lisp.

3.2 Mistral

Mistral [7] è una libreria per risoluzione di vincoli scritta in C++, ha vintro in tre categorie nell'ultima CSC tenutasi nel 2009. L'idea iniziale dello sviluppatore, Emmanuel Hebrard, fu quella di scrivere un risolutore di vincoli leggero, implementando l'algoritmo **MAC** (*Mantaining Arc Consistency*) così come le usuali tecniche che hanno fatto il successo della programmazione con vincoli (euristiche di ordinamento di variabili e valori, vincoli globali ecc.). Nel tempo sono state aggiunte funzionalità come la ricerca branch & bound e con riavvio. Mistral è una libreria, non un linguaggio, infatti offre limitate facilitazioni nella modellazione e zucchero sintattico. La ricerca in Mistral viene effettuata attraverso due algoritmi: il **Binary Backtrack** e una versione leggermente modificata di **AC-3**. Per la CSC del 2009 è stata presentata una versione di Mistral leggermente adattata, i valori da assegnare alle variabili durante la ricerca vengono scelti in ordine lessicografico o random, mentre per le variabili è stata implementata una versione modificata dell'euristica del dominio diviso il grado pesato di una variabile spesso chiamata *dom/wdeg*, il cambiamento riguarda l'aumento del peso associato al vincolo della variabile esaminata quando l'algoritmo di ricerca incontra un fallimento. Questo peso viene incrementato della differenza tra il massimo livello calcolabile dell'albero di ricerca e il livello in cui si incontra il fallimento, anzichè di 1. L'intuizione dietro questa scelta è che un fallimento incontrato in una fase iniziale della ricerca è più significativo di un fallimento più tardivo. Oltre alle euristiche anche la politica di riavvio ha subito una modifica, in una delle versioni di Mistral presentate alla competizione è implementata una politica di riavvio geometrica. Il valore iniziale di

riavvio è settato a 2/3 del livello massimo dell'albero e viene moltiplicato di 1 + 1/3 ad ogni riavvio.

3.3 Abscon

Abscon [13] è un risolutore di vincoli scritto in Java che partecipa alla CSC dal 2006, sviluppato al centro CRIL-CNRS dell'Università di Artois. Abscon utilizza tecniche molto simili a quelle utilizzate in Mistral come l'algoritmo **MAC**, esplora lo spazio di ricerca con euristiche come la *dom/wdeg* e usa il restart della computazione dopo aver raggiunto un *cutoff* (come per esempio il numero di backtracks durante una ricerca). Un'altra caratteristica di Abscon sono le "Transposition Tables", un approccio utile a tagliare rami dell'albero di ricerca che possono essere ignorati. Abscon si è classificato ai primi posti in quasi tutte le categorie della CSC del 2009, risultando quindi un buon risolutore da utilizzare.

3.4 Formalismi di modellazione CSP

La programmazione con vincoli ha attirato grande attenzione tra gli esperti di molte aree grazie al suo potenziale nel risolvere problemi reali e alle sue salde basi teoriche. Nonostante ciò manca una rappresentazione standardizzata per la rappresentazione di istanze di problemi che limita l'accettazione della programmazione con vincoli dal mondo commerciale. Due linguaggi che perseguono questo scopo meritano citazione: FlatZinc e XCSP. Il primo fu creato originariamente per essere il linguaggio target in cui tradurre di istanze CSP di alto livello scritte in MiniZinc [27], un altro linguaggio per modellare CSP di più alto livello rispetto a FlatZinc, ed essere facile da trasformare in una forma comprensibile dai risolutori. Oggi FlatZinc viene principalmente usato

come lingua franca per la valutazione e testing di risolutori. Per esempio, dal 2008 FlatZinc è stato usato nella MiniZinc Challenge, una competizione in cui differenti risolutori sono confrontati usando un benchmark di istanze MiniZinc compilate in FlatZinc.

XCSP è strutturalmente simile a FlatZinc. Proposto inizialmente alla CSC 2005 per i risolutori in competizione, è stato usato ed esteso in altri contesti. La necessità di uno standard nel rappresentare istanze di CSP è anche causata dal grande numero di risolutori in circolazione e dalla loro diversità. Attualmente solo pochi risolutori supportano nativamente sia FlatZinc che XCSP. Sfortunatamente, Gecode, uno dei risolutori più conosciuti e utilizzati, supporta nativamente solo FlatZinc. Per questa ragione è stato creato *x4g*, un plug-in che permette di risolvere problemi definiti in XCSP usando Gecode. L'obiettivo di *x4g* è duplice, per prima cosa si è voluto sfruttare Gecode per risolvere problemi in XCSP senza considerare dettagli implementativi di basso livello e senza scrivere una singola linea di codice C++; secondariamente si è voluto fornire alla comunità della programmazione con vincoli un tool per valutare le performance di Gecode rispetto agli altri risolutori partecipanti alla CSC. Questo può essere molto interessante in quanto, al meglio delle nostre conoscenze, il benchmark usato alla CSC è il più vasto disponibile per numero e varietà di istanze contenute nel mondo della programmazione con vincoli.

3.5 XCSP

XCSP [24] è un formalismo per rappresentare CSP usando l'XML, diventato il formato standard nella CSC. XCSP è stato definito con lo scopo di essere il modello di riferimento per tutti i risolutori di CSP. L'obiettivo di una rappresentazione XML è di facilitare il testing di diversi algoritmi fornendo un comune banco di prova di istanze CSP. La rappresentazione presentata è di

basso livello: ogni istanza di dominio, variabile, relazione (se esistente), predicato (se esistente) e vincolo è definita esaustivamente. Non esistono strutture di controllo come il ciclo "for" o l'istruzione "if then else".

XCSP è presentato in due varianti: la *fully-tagged* è puro XML completamente gerarchizzato e strutturato, adatto all'utilizzo di tool XML generici ma più verboso e difficile da scrivere per un essere umano; la notazione *abridged* è un'abbreviazione della *fully-tagged*, facilmente leggibile e scrivibile da un essere umano ma difficilmente gestibile da un tool XML. L'obiettivo degli sviluppatori è stato quello di raggiungere una rappresentazione:

- leggibile: grazie all'XML è possibile modificare a mano elementi di un'istanza CSP;
- concisa: nella versione *abridged* le espressioni possono essere espresse in maniera più compatta, comparabile a una rappresentazione tabellare;
- strutturata: perchè il formato è basato sull' XML, rimane piuttosto facile effettuare parsing di istanze XCSP.

Come esempio di istanza XCSP si può considerare la seguente dove il ben conosciuto vincolo "all different" è applicato a due variabili A_1 e A_2 che possono assumere valori solo nel dominio [1,2].

```
<domains nbDomains="1">
  <domain name="d0" nbValues="2">1..2</domain>
</domains>
<variables nbVariables="2">
  <variable name="A1" domain="d0"/>
  <variable name="A2" domain="d0"/>
</variables>
<constraint name="c0" arity="2" scope="A1 A2"
 reference="global:alldifferent"/>
```

3.6 X4G

In principio la traduzione di istanze XCSP in Gecode è similare al funzionamento del plug-in di Gecode che effettua il parsing di istanze FlatZinc e produce una struttura dati interna (Gecode Space Object) che viene poi usata per trovare la soluzione del CSP. Per usare Gecode come risolutore di CSP definiti in XCSP, si potrebbe pensare di implementare un compilatore da XCSP a FlatZinc, questo approccio potrebbe però portare perdita di informazioni. Si è deciso quindi di creare una traduzione diretta da XCSP a Gecode, guadagnando così flessibilità e indipendenza dal plug-in per FlatZinc. Per lo sviluppo del plug-in *x4g* è stato usato il parser per XCSP fornito dagli organizzatori della CSC. Questo parser, sviluppato in particolare per supportare la notazione *abridged*, è scritto in C++, facendo uso delle conosciute librerie libxml2. Ogni vincolo contenuto nel file XCSP genera un equivalente numero di vincoli di Gecode, quando tutti i vincoli XCSP sono tradotti in vincoli Gecode, un Gecode Space Object viene ritornato. Questo oggetto può essere usato successivamente per trovare la soluzione del CSP usando una delle strategie di ricerca predefinite di Gecode. XCSP prevede di esprimere vincoli su variabili a valore intero; le variabili devono essere preventivamente dichiarate associandole, ognuna, al proprio dominio. Prevede quindi una sezione `domains`, dove vengono definiti i singoli `domain` come intervalli di valori interi o come insieme di singoli interi e/o intervalli, e una sezione `variables`, dove vengono definite le singole `variable` associandole a un ben preciso dominio. Ovviamente nulla impedisce di associare più variabili allo stesso dominio. La conversione in Gecode di questo meccanismo di domini/variabili è stata immediata e sostanzialmente ovvia: parsando la sezione `domains` si prepara un vettore di `Gecode::IntSet` (struttura dati per rappresentare insiemi di interi) dove ogni elemento rappresenta un dominio. Quando poi il parser incontra la definizione delle singole variabili, con l'indicazione sia del nome che dell'indice del relativo dominio, la

corrispondente `Gecode::IntVar` viene creata con costruttore che gli associa il relativo `IntSet`. La versione base di XCSP prevede tre tipi di vincoli: *relations*, *predicates*, *global constraints*. I primi, prima di essere utilizzati, devono essere definiti in una apposita sezione (un elemento `relations`, contenitore di singoli elementi `relation`); analogamente i secondi (un elemento `predicates`, contenitore di singoli elementi `predicate`). I `global constraints` invece sono definiti (con un'unica eccezione) a priori in un catalogo esterno. La variante CSP di XCSP prevede due tipi (i `semantics`) di `relations`: `supports` e `conflicts`. In entrambi i casi le relazioni sono espresse come liste di tuple di valori interi di cardinalità costante. In entrambi i casi devono essere invocate con un numero di parametri (esclusivamente variabili, contrariamente ai `predicates`) pari alla cardinalità delle tuple. Il significato tra le due tipologie è però profondamente diverso. Le `supports` sono liste di tuple di combinazioni di vali valori ammessi per le variabili su cui si esprimono. Le `conflicts` sono invece liste di tuple di combinazioni di vali valori non ammessi per le variabili su cui si esprimono. Si è usato il vincolo di disuguaglianza per tradurre queste relazioni in Gecode. I `predicate` sono espressioni parametriche con risultato booleano che corrispondono a vincoli che sono considerati soddisfatti quando l'espressione assume il valore `true`. Vengono invocate con un numero di parametri prefissato (specifico di ogni predicato) ma, contrariamente alle relazioni, i parametri passati possono anche essere valori interi. Anche per i predicati la conversione in Gecode è stata diretta, infatti Gecode ha per quasi tutti i predicati di XCSP un API in grado di postare l'equivalente vincolo. Per quanto riguarda i `global constraints`, XCSP supporta la maggior parte dei `global constraints` definiti nel Global Constraint Catalog, questo catalogo contiene centinaia di `global constraints`. In *x4g* si è deciso di implementare un sottoinsieme dei più usati `global Constraints`, scegliendo i seguenti: `alldifferent`, `among`, `atleast`, `atmost`, `cumulative`, `diffn`, `disjunctive` , `element`, `global cardinality`, `lex less`, `lex lesseq`, `not`

`all equal`, `weightedSum`. Sfortunatamente, il parser supporta solo un limitato numero di `global constraints`, rendendone necessaria la modifica per disporre dei necessari. Il formato XCSP è principalmente usato per specificare CSP, ma supporta anche estensioni per definire vincoli pesati o quantificatori sui vincoli. *x4g* è stato scritto con l'obiettivo di accettare solo CSP, non gestisce quindi queste funzionalità addizionali.

Capitolo 4

Raccolta dati

Per acquisire una discreta quantità di dati sulle prestazioni dei solver si è scelto di attingere al benchmark più esteso nel mondo della programmazione con vincoli, ovvero al benchmark di CSP in formato XCSP della CSC, seguendo le regole della competizione per quanto riguarda il formato della risposta e il tempo massimo di calcolo per ogni problema fissato a 1800 secondi. Per eseguire i test sono state utilizzate macchine con le seguenti caratteristiche: CPU Intel Core 2 Duo E7500 2.93GHz con 3072 KB di cache, RAM 2 GB, 32 bit di indirizzamento. I risolutori Mistral e Abscon sono stati utilizzati con le impostazioni presentate alla CSC. Gecode invece, non avendo mai partecipato alla competizione a causa del mancato supporto del formato XCSP, è stato testato in varie versioni, in totale 15, con le principali euristiche e tecniche di ricerca. In particolare, le euristiche di scelta delle variabili scelte sono:

- Variabile con dominio minimo.
  (INT_VAR_SIZE_MIN).

- Variabile con numero di propagatori di vincoli massimo.
  (INT_VAR_DEGREE_MAX).

- Variabile con AFC massimo.
  (INT_VAR_AFC_MAX).

- Variabile con cardinalità del dominio diviso l'AFC rispettivamente minimo/massimo.
  (INT_VAR_SIZE_AFC_MIN/MAX).

- Variabile con cardinalità del dominimo diviso il numero di propagatori dipendenti dalla variabile rispettivamente minimo/massimo.
  (INT_VAR_SIZE_DEGREE_MIN/MAX).

più varie combinazioni delle precedenti sfruttando il *tie-breaking*. Invece le tecniche di ricerca prese in esame sono la DFS e Restart (vedi sezione 3.1).

Per dare un'idea della notevole quantità di calcolo effettuato si consideri ancora che sono state sottoposte ad ogni solver (Mistral, Abscon più le varie versioni di Gecode) tutte le 3293 istanze XCSP della CSC, cioè più di 55000 problemi esaminati in totale, con tempi compresi tra i pochi millisecondi ai 1800 secondi di tempo massimo, per un totale di circa 13000 ore di calcolo. Anche potendo usufruire dai 18 ai 36 calcolatori si è reso indispensabile una progettazione dei test che li rendesse il più possibile flessibili ed autonomi.

4.1 Processi di test

La soluzione migliore per ridurre al minimo i tempi di inattività dei processi di test e l'intervento umano durante il loro svolgimento è sembrata quella di una semplice ma efficace architettura Client-Server su memoria condivisa. Ovviamente sono stati presi in considerazione fattori come: numero di processi attivi contemporaneamente (un processo di test per calcolatore), architettura dei calcolatori (cluster universitario con memoria condivisa), e tempi di implementazione accettabili. Nel dettaglio questa architettura è formata da due

processi simpaticamente chiamati il *Worker* e il *Boss*. Il funzionamento è il seguente: il *Worker*, in una cartella di richieste, crea un file vuoto il cui nome è il nome della macchina o il suo indirizzo di rete (per non avere conflitti) e attende che su questo file venga scritta una riga. Il *Boss* a sua volta legge una riga alla volta da un file contenente tutti i nomi di file da analizzare e scrive su ogni file vuoto nella cartella delle richieste un nome di istanza XCSP. A questo punto i *Worker* troveranno nel loro file una stringa che corrisponde ad un file da analizzare e lanciano il risolutore di vincoli dandogli in input la stringa letta dal file. Finita l'elaborazione e salvato il risultato ogni *Worker* svuota il suo file e attende nuovamente dal *Boss* un nuovo compito. Quando il *Boss* è giunto alla fine del file contenente la lista delle istanze XCSP da elaborare "muore", proprio come il *Worker* quando per qualche ciclo di lettura trova sempre il suo file vuoto. Riassumendo, la comunicazione tra processi è effettuata tramite la scrittura e lettura su file in una cartella condivisa tra tutti i processi. Per non creare conflitti tra i processi di test, ogni processo nomina il suo file con il nome o l'indirizzo di rete della macchina sulla quale viene eseguito e non ci possono essere problemi di scrittura contemporanea su file in quanto solo il *Boss* scrive sui file quando questi sono vuoti mentre i *Worker* leggono i file solo quando questi sono pieni e li svuotano al termine dell'elaborazione. Questa architettura garantisce in un sistema con un centinaio di macchine *Worker* tempi di attesa tra un'esecuzione di un test e la successiva brevi e un carico di lavoro uniforme tra tutti i *Worker* per tutta la durata dei test. Inoltre questa progettazione permette di utilizzare questa architettura a tutti i casi in cui più processi devono elaborare numerosi file senza che l'utente si debba preoccupare di lanciare ogni processo singolarmente.

4.2 Estrazione delle features

Per predire quale solver potrebbe essere più adatto a risolvere un problema si ha bisogno, oltre che del tempo che i solver impiegano a trovare la soluzione, di estrarre le caratteristiche usate per descrivere determinati aspetti di un CSP rispetto ad un altro, così da poter associare al vettore delle features di un problema il solver più efficiente nel risolverlo; queste caratteristiche vengono altresì dette features. Sono state quindi estratte dalle istanze XCSP che compongono la nostra base di test 40 features, utilizzando una parte di CPHydra [6], un portfolio di constraint solver sviluppato presso la University College of Cork (UCC). La scelta delle features ha un impatto significante sulle performance dei classificatori. Se una feature esprime la complessità del problema, come l'arietà di un vincolo, può influenzare l'algoritmo di apprendimento automatico per effettuare una classificazione migliore. L'utilizzo delle features di CPHydra è basato su lavori precedenti [25] che ne hanno testato la bontà.

4.3 Creazione del portfolio tramite classificazione

La fase successiva del lavoro ha coinvolto l'apprendimento automatico. Utilizzando le API di WEKA [8] per la classificazione è stato possibile ottenere una lista di istanze ognuna associata al solver predetto come migliore. Ovviamente più il classificatore è accurato più la previsione si avvicinerà alla migliore possibile. Negli algoritmi di classificazione i dati in input, chiamati anche training set, consistono in records ognuno dei quali avente attributi o caratteristiche multiple. Inoltre ogni record è etichettato con una speciale etichetta di classe. Lo schema in figura 4.1 mostra la struttura di un generico training set.

Nel nostro caso i records sono composti dalle features come attributi e dal

	ATTRIB. 1(A_1)	ATTRIB. 2 (A_2)	.....	ATTRIB. p (A_p)	CLASSE
Record 1					
Record 2					
Record 3					
.....					
.....					
Record n					

Figura 4.1: Struttura generica di un training set.

vettore dei solver migliori per ogni istanza come classi. Vari algoritmi di classificazione implementati in WEKA sono stati presi in esame, ognuno è stato testato sul dataset attraverso le tecniche di cross-validation. La cross-validation è una tecnica statistica utilizzabile in presenza di una buona numerosità del training set. In particolare la *k-fold cross-validation* consiste nella suddivisione del dataset totale in k parti (si chiama anche *k-fold validation*) e, ad ogni passo, la parte $(1/k)$-esima del dataset viene ad essere il validation dataset, mentre la restante parte costituisce il training dataset. Così, per ognuna delle k parti si allena il modello, evitando quindi problemi di overfitting, ma anche di campionamento asimmetrico (e quindi affetto da bias) del training dataset, tipico della suddivisione del dataset in due sole parti (ovvero training e validation dataset). In questo modo è stato anche possibile ottenere un portfolio di solver sulle istanze XCSP, dividendo preventivamente il dataset in 5 parti e usandone a rotazione 4 per il training e 1 per la validazione, i 5 risultati finali sono poi stati ricombinati per ottenere un modello completo. Nel dataset utilizzato sono state preventivamente estratte le istanze che hanno reggiunto il timeout con tutti i solver, questa scelta è dettata dal fatto che un'istanza che non ha nessun solver come migliore non può contribuire alla costruzione del portfolio, non portando effettivamente nessuna informazione su quale solver è più adatto a risolverla. Dei bugs nel codice di calcolo delle features e nel parsing di file XCSP

di Abscon ci hanno costretto a rimuovere altre istanze al dataset restringendolo a circa 2700 istanze. Il modello restituito dal classificatore conteneva i nomi delle istanze XCSP associate al solver predetto come migliore, una volta creato questo modello si è poi rieseguito un test con la stessa architettura dei test eseguiti per esaminare i solver, lanciando però ogni istanza sul solver specificato nel modello.

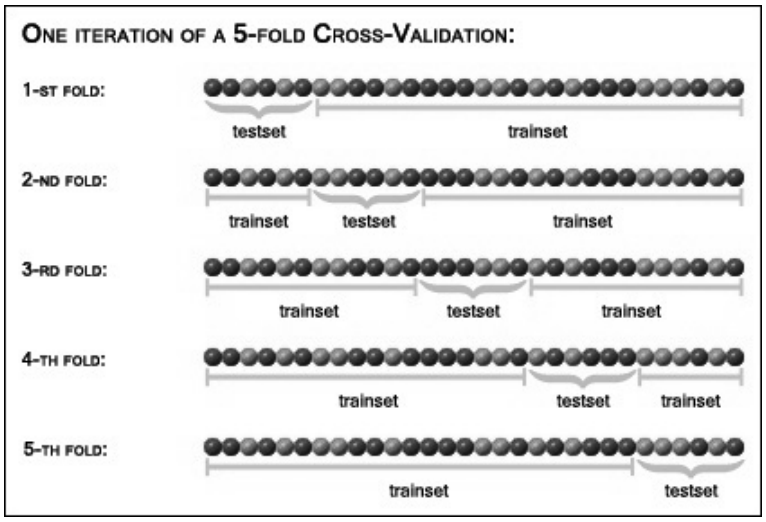

Figura 4.2: Un esempio di cross-validation.

Come algoritmo di classificazione finale è stato scelto il *RandomCommittee*, che confrontato con gli altri algoritmi di classificazione più utilizzati ha ottenuto il numero maggiore di istanze correttamente classificate, cioè le istanze per le quali è stato predetto il solver effettivamente più veloce. In tabella 5.2 sono riportati i dati sulla accuratezza dei migliori classificatori provati. Inoltre viene mostrata la *k-statistic* [10], una misura dell'accordo (coefficient of agreement) tra le risposte qualitative o categoriali di due valutatori oppure dello stesso valutatore in momenti differenti (previsione e osservazione), valutando gli stessi oggetti. Si rimanda alla documentazione ufficiale del software WEKA [8] per una descrizione dettagliata del funzionamento di ogni algoritmo.

4.4 Creazione del portfolio tramite case based reasoning

CPHydra [6] è un portfolio di sover sviluppato al 4C (Cork Constraint Computation Centre) all'University College of Cork (UCC), è stato complessivamente il vincitore della CSC del 2008. Il funzionamento di CPHydra è visibile in figura 4.3. La sua forza risiede nella combinazione di una metodologia di apprendimento automatico chiamata case based reasoning (CBR) [9] con l'idea di partizionare il tempo di CPU tra i componenti del portfolio allo scopo di massimizzare il numero previsto di problemi risolti entro un tempo limite fissato. Il CBR permette di elaborare soluzioni di problemi mai incontrati basandosi sulle similitudini di questi rispetto a casi già affrontati e risolti.

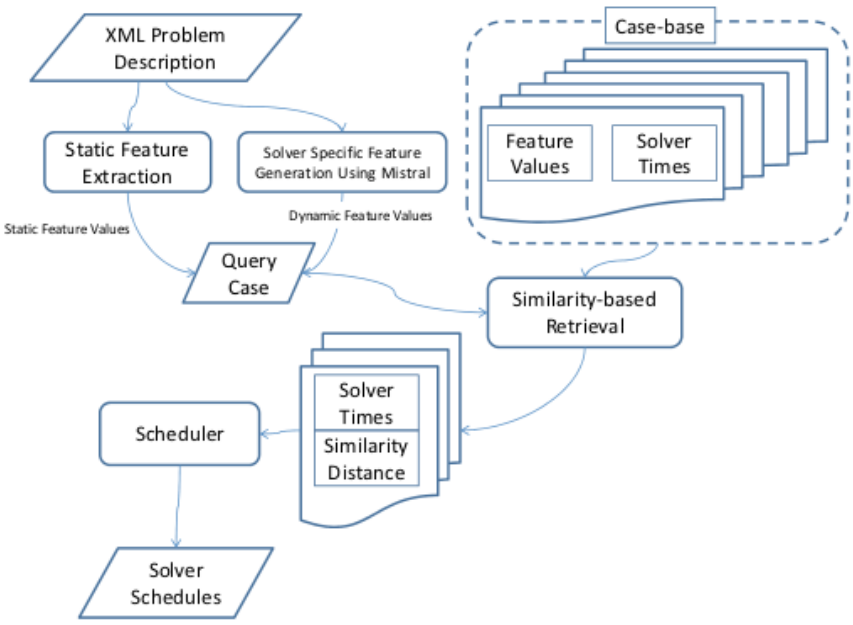

Figura 4.3: Panoramica sul funzionamento di CPHydra.

Le attività principali di un algoritmo CBR possono essere riassunte nella seguente lista:

- **Retrieve**, recupero dei casi più simili;
- **Reuse**, riutilizzo delle informazioni e della conoscenza sui casi più simili per risolvere il problema;
- **Revise**, revisione della soluzione proposta;
- **Retain**, conservazione delle parti dell'esperienza utili ad incrementare il database dei casi per uso futuro.

In figura 4.4 è rappresentato il processo ciclico del CBR. Nella fase di **Retrieve**, CPHydra usa l'algoritmo **k-nearest neighbour** [11] che ritorna l'insieme dei tempi di risoluzione dei k problemi più simili, rispetto alle features, al problema sottomesso. La similitudine dei casi viene calcolata con la classica distanza Euclidea tra le features dei problemi. Durante la fase di **Reuse** i runtime dei k casi più simili sono usati per generare uno schedule dei solver. Per ogni problema sottomesso, y sono i casi simili ritornati, dato solver s e un tempo $t \in [0...1800]$, definiamo $C(s,t)$ come il sottoinsieme di y risolto da s in almeno t tempo. La finalità dello scheduler è allora di massimizzare $C(s,t)$, per ogni solver. Questo risultato è poi ottimizzato pesando il caso in input con la distanza Euclidea con ogni caso simile. Durante la fase di **Revision** una soluzione è valutata e validata avviando ogni solver per l'ammontare del tempo deciso dallo scheduler. Se solo un solver riesce a trovare la soluzione del problema nel tempo dato, allora lo schedule ha avuto successo. Una revisione della soluzione non è possibile in uno scenario competitivo, dove non si avrebbe abbastanza tempo per eseguire ogni solver. Anche per la fase di **Retain**, non si avrebbe abbastanza tempo per costruire un completo nuovo caso, che richiederebbe di conoscere i tempi che tutti i solver del portfolio impiegano a trovare la soluzione del caso.

Anche in questo caso, come per la costruzione del portfolio con tecniche di classificazione, si è suddiviso il dataset (questa volta composto dalle features di

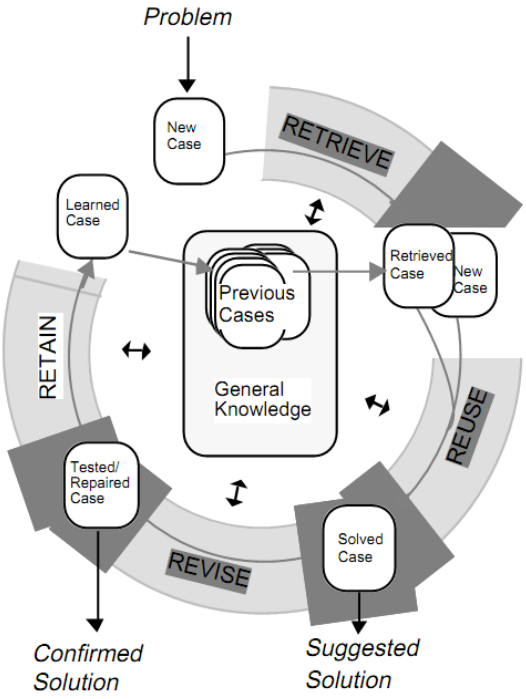

Figura 4.4: CBR: il processo ciclico.

tutte le istanze e dai runtime di tutti i solver) in 5 parti e utilizzato ad ogni ciclo 4 parti per costruire in database dei casi e 1 parte per la costruzione e la simulazione dello scheduler.

Capitolo 5

Risultati

In questo capitolo presentiamo i risultati del lavoro svolto, per ogni fase verranno mostrati grafici e diagrammi riepilogativi e discussi alcuni aspetti dell'analisi dei dati.

5.1 Risultati dei solver

I risolutori di vincoli Gecode, Mistral e Abscon sono stati testati sulle istanze CSP in formato XCSP della Constrain Solver Competition. Mentre Mistral e Abscon, avendo già partecipato alla competizione, supportano XCSP come formato di rappresentazione di un CSP, Gecode ha richiesto un pulg-in: *x4g*; sviluppato in precedentemente ma mai testato in maniera esaustiva su una grande quantità di problemi.

In tabella 5.1 sono riportati i risultati dei solver in termini di numero di problemi di cui è stata trovata una soluzione (SATISFIABLE), numero di problemi di cui non è stata trovata soluzione (UNSATISFIABLE) e numero di problemi di cui dopo il tempo limite fissato a 1800 secondi non si è ancora conclusa l'elaborazione (TIMEOUT). Di tutte le versioni di Gecode viene mostrata solo la versione che ha raggiunto i migliori risultati: ricerca con **Restart**,

euristica di scelta di variabile con *tiebreak* tra INT_VAR_SIZE_AFC_MIN e INT_VAR_DEGREE_MAX.

Solver	SATISFIABLE	UNSATISFIABLE	TIMEOUT
Mistral	1673	1058	561
Abscon	1417	957	720
Gecode	1243	768	1281

Tabella 5.1: Risultati dei solver in termini di numero di istanze risolte, non risolvibili e in timeout.

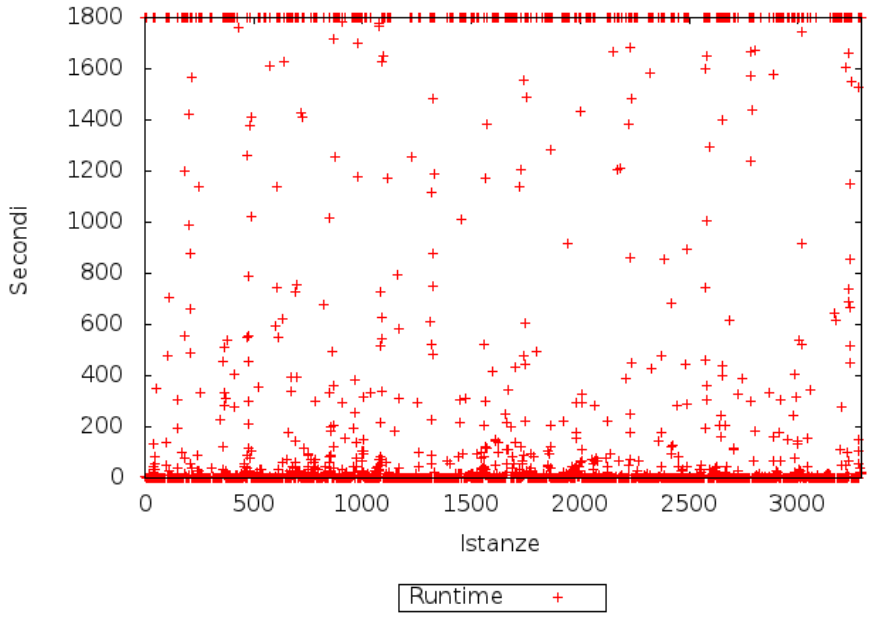

Figura 5.1: Distribuzione dei runtime con il solver Mistral.

E' chiara la supremazia di Mistral, non a caso uno dei migliori in circolazione, sia considerando il tempo totale di calcolo, sia i numero totale di istanze risolte (per risolte si intende le istanze non andate in TIMEOUT). Non è stato possibile testare Abscon su tutte le istanze XCSP che compongono il benchmark a causa

di bugs nel parsing di circa 200 istanze XCSP, comunque contando già 720 istanze andante in TIMEOUT contro le 561 di Mistral, anche considerando le istanze mancanti non avrebbe potuto raggiungere complessivamente i risultati di Mistral. Risultati così scarsi di Gecode non erano invece previsti, essendo il vincitore degli ultimi MiniZinc Challenge in tutte le categorie.

Nelle figure 5.1, 5.2 e 5.3, è raffigurata la distribuzione dei runtime dei solver in tabella 5.1. Anche questa volta Gecode si comporta in maniera imprevista: la distribuzione dei tempi è divisa in due parti, i problemi facili e medi sono abbastanza omogenei, mentre i problemi difficili sembrano irrisolvibili per Gecode, infatti si trovano solamente due problemi risolti tra i 1000 e i 1800 secondi, tutti gli altri sono risultati irrisolvibili nel tempo limite e si concentrano tutti a ordinata 1800.

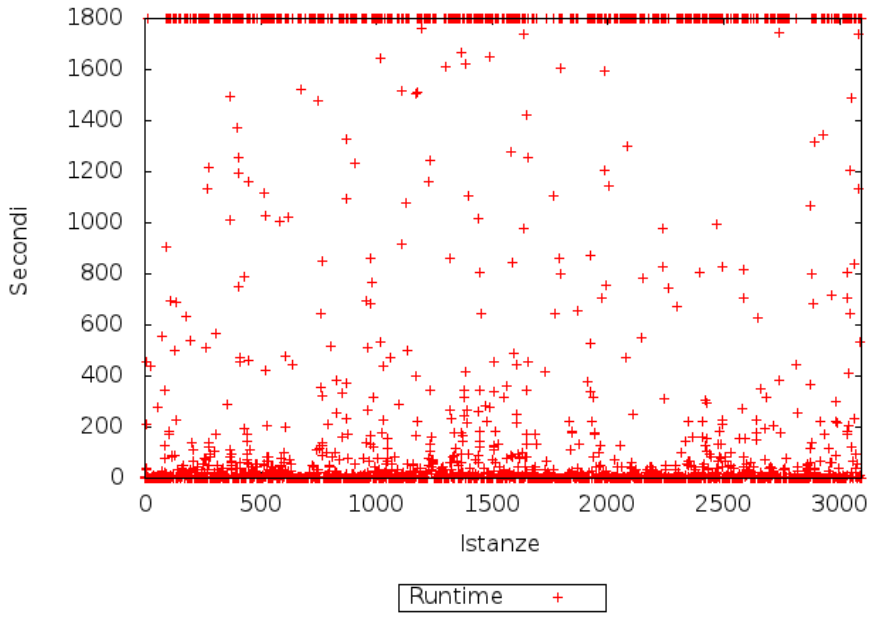

Figura 5.2: Distribuzione dei runtime con il solver Abscon.

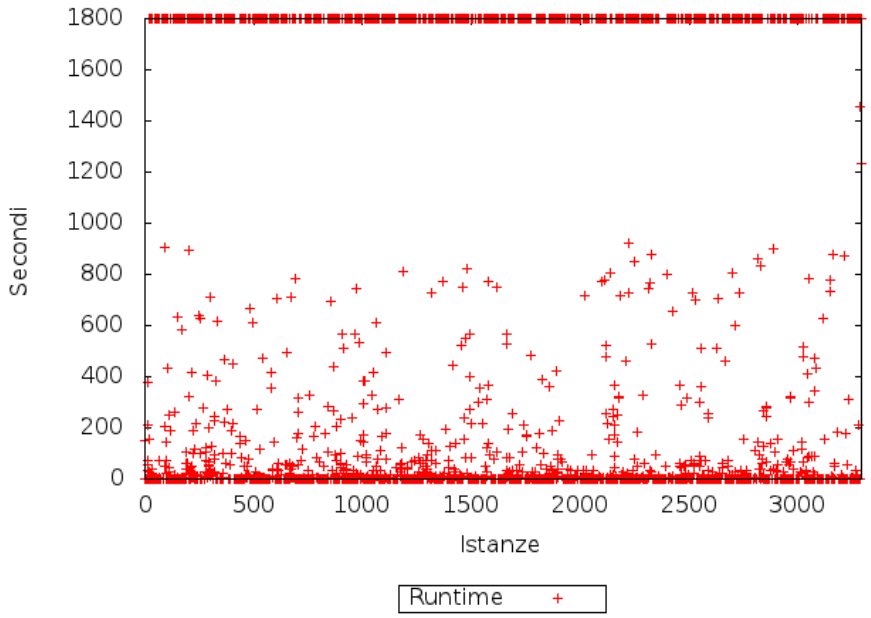

Figura 5.3: Distribuzione dei runtime con il solver Gecode.

5.2 Risultati del portfolio

Verranno qui presentati i risultati del portfolio di solver creato con tecniche di classificazione e case based reasoning. Il dataset, come già spiegato in sezione 5.3, è ristretto a circa 2700 istanze di problemi CSP. Il portfolio di solver virtualmente migliore, cioè quello che verrebbe creato se la classificazione avesse un'accuratezza del 100% verrà chiamato portfolio *perfetto*, mentre il portfolio di solver creato dalla classificazione reale verrà chiamato portfolio *effettivo*.

Due versioni di portfolio sono state costruite a partire dal dataset, nella prima versione abbiamo preso in considerazione tutti i solver (Mistral, Abscon, più tutte le versioni di Gecode testate), ottenendo quindi 17 classi da associare ai vettori delle features. L'accuratezza della classificazione ha risentito ovviamente di questa grande quantità di classi, arrivando nel caso migliore al 80% circa come mostrato in tabella 5.2. Osservando i dati dei vari solver ci si è

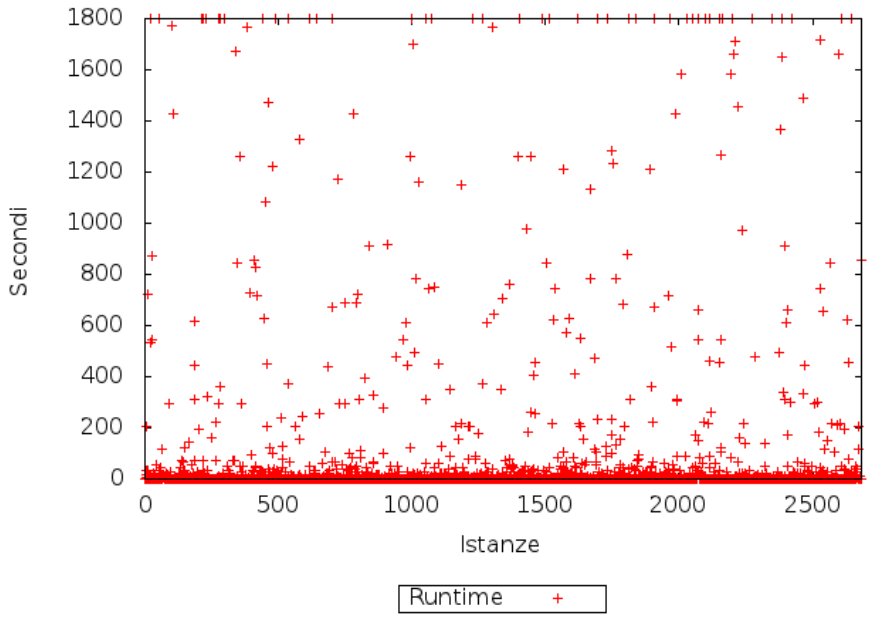

Figura 5.4: Distribuzione dei runtime del portfolio effettivo basato su 3 solver.

però resi conto che le varie versioni di Gecode erano molto simili per risultati di runtime, ovvero, anche se tutte le versioni erano presenti nel portfolio perfetto, in realtà la differenza tra i loro tempi molto spesso era minima. Si è quindi pensato di creare una seconda versione del portfolio considerando solo i solver Mistral, Abscon e una sola versione di Gecode, con ricerca **DFS** ed euristica INT_VAR_SIZE_DEGREE_MIN, che otteneva tempi migliori sulle istanze in cui Abscon e soprattutto Mistral andavano in TIMEOUT. In entrambe le versioni del portfolio, il dataset risentiva del problema dello sbilanciamento delle classi, evento che si verifica quando nel dataset le istanze di alcune classi sono molto meno numerose delle altre. Nel nostro caso lo sbilanciamento si presentava nel caso del portfolio a 3 solver con Mistral, Gecode e Abscon rispettivamente al 61%, 28% e 10%, mentre nel dataset del portfolio a 17 solver Mistral era presente come classe del 48% delle istanze, Abscon del 9,5% e le 15 versioni di Gecode si

spartivano in maniera più o meno omogenea il restante 42,5%. Per bilanciare le classi il dataset è stato quindi preprocessato con i cosiddetti `Filters` di WEKA contenuti nel pacchetto `weka.filters`. Questo pacchetto riguarda classi che trasformano dataset rimuovendo o aggiungendo attributi, ricampionando il dataset ecc. Sperimentalmente il migliore tra i `Filter` è risultato il `Resample` [15].

In figura 5.4 rappresentiamo la distribuzione dei runtime del portfolio effettivo creato con Mistral, Abscon e una versione di Gecode. Questa versione del portfolio ha ottenuto tempi migliori rispetto alla versione con 17 solver come classi. Il portfolio a 3 solver ha battuto tutti i solver sia in tempo totale di calcolo sia in numero totale di istanze risolte, addirittura facendo registrare un tempo totale di calcolo inferiore di circa il 47% e risolvendo circa 100 istanze in più di Mistral, il migliore dei solver testati; mentre il portfolio effettivo a 17 solver ha registrato un runtime totale inferiore all'8% rispetto a Mistral risolvendo però meno istanze.

Classificatore	**Accuratezza**	**K-Statistic**
RandomCommittee	91,05%	0,83
RandomForest	90,64%	0,82
J48	86,91%	0,75
SVM	84,64%	0,69

Tabella 5.2: Accuratezza e k-statistic dei classificatori per il portfolio a 3 solver.

La tecnica di creazione del portfolio con case based reasoning ha risentito anch'essa dello sbilanciamento delle classi del dataset. Il valore del parametro k (che indica il numero di "vicini" da prendere in considerazione durante la scelta della classe da assegnare ad un caso) è fondamentale nell'algoritmo **k-nearest neighbour**, in quanto se una classe è molto numerosa rispetto alle altre, una qualsiasi istanza di una classe poco numerosa avrà facilmente come vicini più

Classificatore	**Accuratezza**	**K-Statistic**
RandomCommittee	80,17%	0,73
RandomForest	79,50%	0,72
SVM	72,49%	0,59
J48	69,66%	0,59

Tabella 5.3: Accuratezza e k-statistic dei classificatori per il portfolio a 17 solver.

prossimi casi appartenenti alla classe numerosa rispetto alla propria. Solo con il settaggio del parametro k a 10 e l'utilizzo come casi dei runtime di Mistral, Abscon e la stessa versione di gecode usata nel portfolio con classificazione a 3 solver, si è riusciti ad ottenere tempi totali calcolo leggermente migliori di Mistral e 52 istanze risolte in più. Come ultimo dato presentiamo in figura 5.5 i runtime dell'elaborazione delle istanze XCSP del CSC dei portfoli creati con le varie tecniche descritte precedentemente messi a confronto con i solver Mistral, Abscon e la versione di Gecode che ha ottenuto il runtime complessivo più basso.

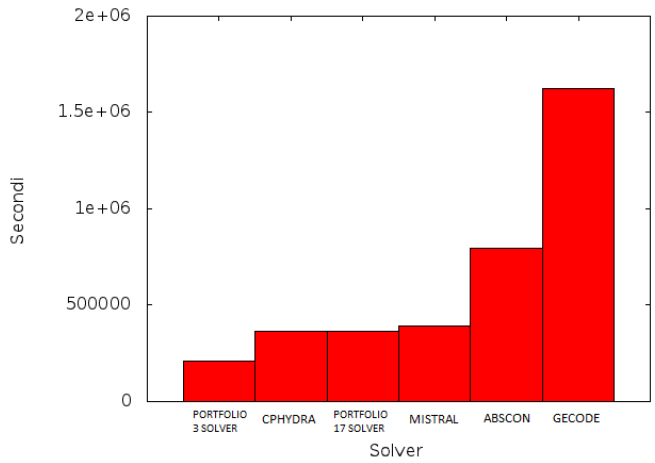

Figura 5.5: Runtime dei portfoli a confronto con i solver testati.

Capitolo 6

Conclusioni

Lo scopo principale di questa tesi è stato inizialmente lo studiare le performance del risolutore di vincoli Gecode su problemi di soddisfacimento di vincoli espressi nel formato XCSP. Precedenti lavori hanno reso possibile l'avvalersi di un plug-in: *x4g*, che traducesse dal formato XCSP alle librerie di Gecode, per la rappresentazione di un CSP. Grazie a *x4g*, si è testato il runtime di Gecode sulle istanze XCSP della Constraint Solver Competition del 2009. Essendo queste istanze molto numerose (quasi 3300) e con grado di complessità (quindi di runtime per la risoluzione) variabile, in una prima fase si è costruita una architettura per i processi di test di tipo client-server autonoma e flessibile, con processi client che ricevono il lavoro da svolgere da un processo server attraverso una comunicazione su memoria condivisa, memorizzando il risultato e aspettando un nuovo compito. Volendo acquisire più informazioni possibili sul runtime di Gecode, esso è stato testato con varie tecniche di ricerca ed euristiche, per un totale di 15 configurazioni. Ottenuti i risultati di Gecode, ci si è spinti oltre confrontandolo con altri due risolutori di vincoli: Mistral e Abscon. Questi, avendo già partecipato alla Constraint Solver Competition supportavano il formato XCSP, non richiedendo nessuna implementazione ulteriore. Risolvendo lo stesso benchmark di istanze XCSP, ci si è resi conto che Gecode richiedeva un

tempo totale molto più alto rispetto a Mistral ed Abscon. Seppur riuscendo a registrare tempi molto buoni nei problemi facili, non era competitivo in quelli difficili superando in questo caso il tempo limite prefissato di 1800 secondi in molte più istanze.

Questo risultato non preclude però il fatto di poter sfruttare ogni solver per la costruzione di un portfolio, ovvero la possibilità di utilizzare per ogni problema XCSP il solver più rapido nel trovarne la soluzione. In questa direzione si è svolta la seconda parte della tesi. Utilizzando i risultati sulle performance dei solver ed altri software come:

- CPHydra per estrarre le informazioni e caratteristiche dei problemi da risolvere oltre che per utilizzarne l'approccio case based reasoning,
- il framework WEKA per servirsi di tecniche di apprendimento automatico, per allenare classificatori che potessero estrarre conoscenza da esempi e utilizzarla per associare ad un insieme di caratteristiche di un problema il solver più adatto a risolverlo,

sono stati costruiti dei solver a portfolio e confrontati con Mistral, Abscon e Gecode.

Il miglior risultato si è conseguito con il portfolio di 3 solver: Mistral, Abscon e una delle versioni di Gecode, creato con classificazione. Questo solver a portfolio ha battuto tutti i solver testati, facendo registrare un runtime totale di quasi il 50% inferiore a quello di Mistral, uno dei più veloci in circolazione e risolvendo più istanze nel tempo limite di ogni altro solver. I risultati, presentati nel capitolo 5, indicano che l'idea di utilizzare un portfolio di solver per risolvere CSP può portare a prestazioni decisamente migliori rispetto all'approccio a singolo solver.

Resta da essere condotto uno studio approfondito sulle tecniche di apprendimento automatico per poter ricavare modelli predittivi ancora più accurati. In-

oltre questo lavoro è stato condotto con l'intenzione di analizzare solo il tempo di calcolo delle soluzioni di un problema XCSP da parte dei solver, la memoria utilizzata è una grandezza che non è stata presa in esame ma può risultare anch'essa decisiva per abbattere il totale di risorse richieste per eseguire questi calcoli.

Bibliografia

[1] Stuart Russell and Peter Norvig. *Intelligenza artificiale. Un approccio moderno*. Prentice Hall, Terza edizione, 2010.

[2] Alan K. Mackworth. *Consistency in networks of relations*. Artif. Intell., 8(1):99–118, 1977.

[3] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, USA, 2006.

[4] Carla P. Gomes and Bart Selman. *Algorithm portfolios. Artif. Intell.*. 126(1-2):43–62, 2001.

[5] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[6] Eoin OMahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry OSullivan. *Using case-based reasoning in an algorithm portfolio for constraint solving*. Proceedings of the 19th Irish Conference on Artificial Intelligence (AICS'08), 2009.

[7] *Pagina web del solver Mistral:* http://homepages.laas.fr/ehebrard/Software.html

[8] *Homepage di Weka 3: Data Mining Software in Java.* http://www.cs.waikato.ac.nz/ml/weka/

[9] Agnar Aamodt and Enric Plaza. *Case-based reasoning: Foundational issues, methodological variations, and system approaches. AI Commun.*. 7(1):39–59, 1994.

[10] J. Cohen. *A coefficient of agreement for nominal scales. Educational and psychological measurement.* 20(1):37, 1960.

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction.* Springer, 2 edition, 2008.

[12] Olivier Roussel and Christophe Lecoutre. *Xml representation of constraint networks: Format xcsp 2.1.* CoRR, abs/0902.2362, 2009.

[13] *Abscon web page:* http://www.cril.univ-artois.fr/ lecoutre/software.html

[14] *Sito web di Gecode:* http://www.gecode.org/

[15] *Documentazione sul filtro WEKA* `Resample` *utilizzato per il prepocessing:* http://weka.sourceforge.net/doc/weka/filters/unsupervised/instance/Resample.html

[16] Massimo Morara, Jacopo Mauro, and Maurizio Gabbrielli, *Solving XCSP problems by using Gecode*, www.sci.unich.it/cilc2011/papers/morara.pdf

[17] Jordan Bell e Brett Stevens, *A survey of known results and research areas for n-queens*, Discrete Mathematics, vol. 309(1)1-31, 2009.

[18] Kenneth Appel, *Wolfgang Haken, John Koch, Every Planar map is Four Colorable*, Illinois Journal of Mathematics, vol. 21:439-567, 1977

[19] A. Colmerauer, Prolog II Reference Manual and Theoretical Model, Internal Report, GroupeIA, U Aix-Marseille, 1982

[20] Peter Van Roy and Seif Haridi, *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004

[21] *GNU Prolog website:* http://www.gprolog.org/

[22] *Sito web della competizione CSC:* http://www.cril.univ-artois.fr/CPAI09/

[23] *ILOG CP Optimizer:* http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/

[24] *XML Representation of Constraint Networks Format XCSP 2.1*, Organising Committee of the Third International Competition of CSP Solvers, 2008

[25] Zeynep Kiziltan, Luca Mandrioli, Jacopo Mauro, and Barry O'Sullivan, *A Classification-based Approach to Manage a Solver Portfolio for CSPs*, The 22nd Irish Conference on Artificial Intelligence and Cognitive Science (AICS-2011), University of Ulster, 2011

[26] *Competizione MiniZinc:* http://www.g12.cs.mu.oz.au/minizinc/challenge2010/

[27] *Specifiche del linguaggio MiniZinc:* http://www.g12.cs.mu.oz.au/minizinc/specifications.html

Ringraziamenti

Ringrazio il Prof. Maurizio Gabbrielli e il Dr. Jacopo Mauro per l'aiuto e il tempo che hanno dedicato a questo lavoro di tesi.

Il più grande ringraziamento va ai miei genitori Filippa Ciminnisi e Antonino Riccardo per essere stati una guida e per il sostegno che non mi hanno mai fatto mancare, e a mio fratello Lorenzo Riccardo per avermi consigliato e motivato durante tutto il percorso universitario.

Un grande ringraziamento va anche a Odeta Qorri per essermi stata vicina ed avermi supportato in tutti i momenti difficili della mia carriera universitaria e della mia vita.